Swift errors represent recoverable situations. Errors belong to any type conforming to the protocol[2] `Error`. Functions that may throw errors—known as **throwing functions**—must declare this fact by placing the keyword `throws` (see `second` and `third` in the example above) after its parameter list. Calls to throwing functions must be marked with `try`. Errors are caught in a `do-catch` statement; uncaught errors are propagated to the surrounding scope. You can, if desired, convert an error to an optional by invoking `try? f()`, producing the (wrapped) result of the call, if any, or `nil` if the f throws.

12.4 OPERATORS

Just as Swift does not bake in its basic types but instead defines them in its standard library, so too does it define operators as regular functions within the library. The standard library defines a rich set of operators, including the prefix operators `!` (logical not), `~` (bitwise not), `+` (unary plus), and `-` (unary negation). The binary operators, from highest to lowest precedence, follow:

Operator(s)	**PrecedenceGroup**	**Asc**	**Description**
`<<  >>`	BitwiseShift	None	left shift, right shift

`*  /  %`
`&*  &`
	Multiplication	L	multiply, divide, remainder, multiply w/overflow, bit-and

`+  -  &+`
`&-  |  ^`
	Addition	L	add, subtract, add w/overflow, subtract w/overflow, bit-or, bit-xor

`..<  ...`	Range	None	half-open range, closed range

`is`
`as as? as!`
	Casting	L	type check, cast, cast as optional, unwrap cast

`??`	NilCoalescing	R	nil coalesce

`<  <=  > >=`
`=  !=`
`===  !==`
`~=`
	Comparison	None	less, less or equal, greater, greater or equal, equal, not equal, identical, not identical, pattern match

`&&`	LogicalConjunction	L	(short-circuit) logical and
`||`	LogicalDisjunction	L	(short-circuit) logical or
`?:`	Ternary	R	conditional

`=  *=  /=  %=`
`+=  -=  <<=  >>=`
`&=  |=  ^=`
`&&=  ||=`
	Assignment	R	assignment

As in Elm, Swift operators are just functions, so you can define your own implementations of them on your own data types. We've done this below in our recurring vectors example, to which we've taken the liberty to add a prefix unary negation operator:

```
import Foundation

struct Vector: CustomStringConvertible {
    let i: Double
    let j: Double

```

[2]We'll get to details of protocols shortly; for now, think of them as interfaces in Java or Go.

```
    func magnititude() -> Double {
        return sqrt(i * i + j * j)
    }

    var description: String {
        return "<\(i),\(j)>"
    }
}

func + (left: Vector, right: Vector) -> Vector {
    return Vector(i: left.i + right.i, j: left.j + right.j)
}

func * (left: Vector, right: Vector) -> Double {
    return left.i * right.i + left.j * right.j
}

prefix func - (v: Vector) -> Vector {
    return Vector(i: -v.i, j: -v.j)
}

let u = Vector(i: 3, j: 4)
let v = Vector(i: -5, j: 10)
assert(u.i == 3)
assert(u.j == 4)
assert(u.magnititude() == 5)
assert(String(describing: u + v) == "<-2.0,14.0>")
assert(u * v == 25)
assert(String(describing: -v) == "<5.0,-10.0>")
```

You can even define your own operators. There are quite a few rules surrounding which characters can appear in these **custom operators** (see [4] for full details). Among these rules: custom operator names must be made up of “symbol”-like characters, operators containing dots (`.`) must begin with a dot, a lone question mark is not an operator, and no postfix operator may begin with a question mark or exclamation mark.

Let’s quickly run through the steps in creating a couple custom operators. We must *declare* the operators before defining them, assigning binary operators to a precedence group (either an existing group, defined in the Swift Standard Library, or one we define ourselves) and assigning an associativity (`left`, `right`, or `none`). Here we’ve created an infix `~|*|~` that binds more tightly than addition but less tightly than multiplication, by first creating a new precedence group:

```
precedencegroup JustForFun {
  higherThan: AdditionPrecedence
  lowerThan: MultiplicationPrecedence
  associativity: left
}
```

```
infix operator ~|*|~ : JustForFun
postfix operator ^^

func ~|*|~ (x: Int, y: Int) -> Int {
    return 2 * x + y
}

postfix func ^^ (x: Int) -> Int {
    return x - 2
}

assert(8^^ ~|*|~ 3 == 15)
```

12.5 PROTOCOLS

Now let's return to Swift's type system. We've seen five of the six kinds of types so far: structures (including numbers booleans, strings, arrays, and dictionaries), enumerations (including optionals), classes, tuples, and functions. The sixth, **protocols**, is Swift's analog of Java's interfaces and Ruby's mixins. A protocol specifies certain requirements that structures, enumerations and classes that wish to **adopt** the protocol must **conform** to.

For our introductory example, we define a protocol `Summarizable` for things with summaries, give a struct and enum that adopt it, and invoke the summary property through variables declared with the protocol type:

```
import Foundation

protocol Summarizable {
    var summary: String { get }
}

struct Circle: Summarizable {
    var radius = 1.0
    var summary: String {return "Circle with radius \(radius)"}
}

enum Direction: Int, Summarizable {
    case north, east, south, west
    var summary: String {return "Bearing \(90 * self.rawValue)"}
}

let a: [Summarizable] = [Circle(radius: 5), Direction.west]
assert(a[0].summary == "Circle with radius 5.0")
assert(a[1].summary == "Bearing 270")
```

The adopted protocols appear in the type declaration, comma separated, following any superclass specification (for classes) or raw type specification (for enums). The protocol definition itself defines a number of requirements for not only properties but initializers and functions (methods) as well.